Docket No.: 42390P11931
Express Mail No. EL651893706US

UNITED STATES PATENT APPLICATION

for

AN APPARATUS AND METHOD FOR UNILATERALLY LOADING A SECURE OPERATING SYSTEM WITHIN A MULTIPROCESSOR ENVIRONMENT

Express Mail Label No. EL651893706US

Inventors:

Michael A. Kozuch
James A. Sutton II
David Grawrock
Gilbert Neiger
Richard Uhlig
Bradley G. Burgess
David I. Poisner
Clifford D. Hall
Andy Glew
Lawrence O. Smith III
Robert T. George

Prepared by:

Blakely, Sokoloff, Taylor & Zafman LLP
12400 Wilshire Boulevard
Seventh Floor
Los Angeles, CA 90025-1026
(310) 207-3800

AN APPARATUS AND METHOD FOR UNILATERALLY LOADING A SECURE OPERATING SYSTEM WITHIN A MULTIPROCESSOR ENVIRONMENT

FIELD OF THE INVENTION

[001] The invention relates generally to the field of computer security. More particularly, the invention relates to a method and apparatus for unilaterally loading a secure operating system within a multiprocessor environment.

BACKGROUND OF THE INVENTION

[002] As computers become more integrated into our society, the need for computer security drastically increases. Recently, Internet commerce has experienced a vast growth over the computer networks of the world. Unfortunately, unless Internet commerce is adequately protected, using full-proof computer security mechanisms, the potential for computer piracy may one day erode consumer confidence. In other words, computer users which provide confidential information in order to acquire products and services must have adequate insurance that the information will not be intercepted by computer pirates.

[003] As a result, many computer systems now incorporate vital security features such as encryption, source verification, trusted environment, as well as additional security features. As such, current online computer systems generally rely on transitive trust relationships. The public key infrastructure (PKI) is an example of such a transitive trust model. Under the public key infrastructure, a certification authority may provide an individual with a private key that only the user is aware of.

[004] Accordingly, when the user provides information, it may be encrypted using the computer user's private key. As such, a recipient of the encrypted information may obtain a public key in order to decrypt the encrypted information by contacting a certification authority. In addition, a source of information may also be authenticated by digitally signing messages, which may also be decrypted in order to verify a source of information.

[005] As one can see, the PKI provides mechanisms which ensure security for one-to-one relationships. However, relationships can quickly grow beyond one-to-one interactions, which require transitive trust to ensure security. Unfortunately, trust is generally not transitive. For example, an individual may trust a certification authority and receive an issued extrinsic certificate from the certification authority. Following issuance of the certificate, the certification authority may decide to trust a further individual and grant the individual access and control to all of the issued certificates, including of course the initial individual's certificate.

[006] Unfortunately, the initial individual may not trust the subsequent individual which is trusted by the certification authority. Accordingly, had the individual known that the certification authority trusted the subsequent individual prior to issuance of the certificate, the individual probably would not have requested the certificate. As such, the problem illustrates that transitive trust is neither symmetric nor transitive nor distributed. In other words, the only reliable trust is self-trust, which cannot have an unknown subsequent individual which is trusted by the certification authority following an initial formation of trust. Although trust of a third party is not always unreliable, it cannot always be reliably estimated.

[007] In some computer systems, the user or system administrator may desire to load a trustable operating system. By trustable, what is meant is that the user, or a third party requires a mechanism for inspecting the system and determining whether a given operating system was loaded. Once verification of loading of the operating system is complete, an outside agent may also desire to determine whether the operating system was loaded in the secure environment. Unfortunately, this capability cannot be supported with conventional transitive trust models, such as the public key infrastructure. Therefore, there remains a need to overcome one or more of the limitations in the above-described existing art.

BRIEF DESCRIPTION OF THE DRAWINGS

[008] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which:

[009] FIG. 1 depicts a block diagram illustrating a network computer environment as known in the art.

[0010] FIG. 2 depicts a block diagram illustrating a conventional computer system.

[0011] FIG. 3 depicts a block diagram illustrating a system wherein the present invention may be practiced in accordance with an embodiment of the present invention.

[0012] FIG. 4 depicts a block diagram illustrating a multiprocessor computer system for loading a trustable operating system in accordance with a further embodiment of the present invention.

[0013] FIGS. 5A and 5B depict block diagrams illustrating secure memory environments in accordance with a further embodiment of the present invention.

[0014] FIG. 6 depicts a flowchart illustrating a method for unilaterally loading a secure operating system within a multiprocessor environment in accordance with one embodiment of the present invention.

[0015] FIG. 7 depicts a flowchart illustrating an additional method for loading an operating system within a memory region which will become a secure memory environment in accordance with an embodiment of the present invention.

[0016] FIG. 8 depicts a flowchart illustrating an additional method for disregarding a received LSR instruction in accordance with the further embodiment of the present invention.

[0017] FIG. 9 depicts a flowchart illustrating an additional method for creating a secure memory environment in accordance with a further embodiment of the present invention.

[0018] FIG. 10 depicts a flowchart illustrating an additional method for completing formation of a secure memory environment in accordance with a further embodiment of the present invention.

[0019] FIG. 11 depicts a flowchart illustrating a method for establishing security verification of the secure memory environment to an outside agent in accordance with a further embodiment of the present invention.

[0020] FIG. 12 depicts a flowchart illustrating a method performed in response to an SRESET command in accordance with one embodiment of the present invention.

[0021] FIG. 13 depicts a flowchart illustrating an additional method for resetting one or more processors in response to receipt of an SRESET command in accordance with the further embodiment of the present invention.

[0022] FIG. 14 depicts a flowchart illustrating an additional method for re-enabling processor read-write access to a secure memory environment formed in accordance with an exemplary embodiment of the present invention.

DETAILED DESCRIPTION

[0023] A method and apparatus for unilaterally loading a secure operating system within a multiprocessor environment are described. The method includes disregarding a received load secure region instruction when a currently active load secure region instruction is detected. Otherwise, a memory protection element is directed, in response to the received load secure region instruction, to form a secure memory environment. Once directed, unauthorized read/write access to one or more protected memory regions of the secure memory environment is prohibited. Finally, a cryptographic hash value of the one or more protected memory regions is stored within a digest information repository. Once stored, outside agents may request access to the encrypted identification information in order to establish security verification of a secure operating system is within the secure environment.

[0024] In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without some of these specific details. In addition, the following description provides examples, and the accompanying drawings show various examples for the purposes of illustration. However, these examples should not be construed in a limiting sense as they are merely intended to provide examples of the present invention rather than to provide an exhaustive list of all possible implementations of the present invention. In other instances, well-known structures and devices are shown in block diagram form in order to avoid obscuring the details of the present invention.

[0025] Portions of the following detailed description may be presented in terms of algorithms and symbolic representations of operations on data bits. These algorithmic descriptions and representations are used by those skilled in the data processing arts to convey the substance of their work to others skilled in the art. An algorithm, as described herein, refers to a self-consistent sequence of acts leading to a desired result. The acts are those requiring physical manipulations of physical quantities. These quantities may take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. Moreover, principally for reasons of common usage, these signals are referred to as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0026] However, these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise, it is appreciated that discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or displaying" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities

within the computer system's devices into other data similarly represented as physical quantities within the computer system devices such as memories, registers or other such information storage, transmission, display devices, or the like.

[0027] The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose systems may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the required method. For example, any of the methods according to the present invention can be implemented in hard-wired circuitry, by programming a general-purpose processor, or by any combination of hardware and software.

[0028] One of skill in the art will immediately appreciate that the invention can be practiced with computer system configurations other than those described below, including hand-held devices, multiprocessor systems, microprocessor-based or programmable consumer electronics, digital signal processing (DSP) devices, network PCs, minicomputers, mainframe computers, and the like. The invention can also be practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. The required structure for a variety of these systems will appear from the description below.

[0029] It is to be understood that various terms and techniques are used by those knowledgeable in the art to describe communications, protocols, applications, implementations, mechanisms, etc. One such technique is the description of an implementation of a technique in terms of an algorithm or mathematical expression. That is, while the technique may be, for example, implemented as executing code on a computer, the expression of that technique may be more aptly and succinctly conveyed and communicated as a formula, algorithm, or mathematical expression.

[0030] Thus, one skilled in the art would recognize a block denoting A+B=C as an additive function whose implementation in hardware and/or software would take two inputs (A and B) and produce a summation output (C). Thus, the use of formula, algorithm, or mathematical expression as descriptions is to be understood as having a physical embodiment in at least hardware and/or software (such as a computer system in which the techniques of the present invention may be practiced as well as implemented as an embodiment).

[0031] In an embodiment, the methods of the present invention are embodied in machine-executable instructions. The instructions can be used to cause a general-purpose or special-purpose processor that is programmed with the instructions to perform the steps of the present invention. Alternatively, the steps of the present invention might be performed by specific hardware components that contain hardwired logic for performing

the steps, or by any combination of programmed computer components and custom hardware components.

[0032] In one embodiment, the present invention may be provided as a computer program product which may include a machine or computer-readable medium having stored thereon instructions which may be used to program a computer (or other electronic devices) to perform a process according to the present invention. The computer-readable medium may include, but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAMs), Erasable Programmable Read-Only Memory (EPROMs), Electrically Erasable Programmable Read-Only Memory (EEPROMs), magnetic or optical cards, flash memory, or the like.

[0033] Accordingly, the computer-readable medium includes any type of media/machine-readable medium suitable for storing electronic instructions. Moreover, the present invention may also be downloaded as a computer program product. As such, the program may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client). The transfer of the program may be by way of data signals embodied in a carrier wave or other propagation medium via a communication link (e.g., a modem, network connection or the like).

System Architecture

[0034] Referring now to FIG. 1, FIG. 1 depicts a network environment 100 in which the techniques of the present invention may be implemented. As shown, the network environment includes several computer systems such as a plurality of servers 104 (104-1, . . ., 104-M) and a plurality of clients 108 (108-1, . . ., 108-N), connected to each other via a network 102. The network 102 may be, for example, the Internet. Note that alternatively the network 102 might be or include one or more of: a Local Area Network (LAN), Wide Area Network (WAN), satellite link, fiber network, cable network, or a combination of these and/or others. The method and apparatus described herein may be applied to essentially any type of communicating means or device whether local or remote, such as a LAN, a WAN, a system bus, a disk drive, storage, etc.

[0035] Referring to FIG. 2, FIG. 2 illustrates a conventional personal computer 200 in block diagram form, which may be representative of any of the clients 108 and servers 104, shown in FIG. 1. The block diagram is a high level conceptual representation and may be implemented in a variety of ways by various architectures. The computer 200 includes a bus system 202, which interconnects a Central Processing Unit (CPU) 204, a Read Only Memory (ROM) 206, a Random Access Memory (RAM) 208, a storage 210, a display 220, an audio 222, a keyboard 224, a pointer 226, miscellaneous input/output (I/O) devices 228, and communications 230.

[0036] The bus system 202 may be for example, one or more of such buses as a system bus, Peripheral Component Interconnect (PCI), Advanced Graphics Port (AGP), Small Computer System Interface (SCSI), FireWire, etc. The CPU 204 may be a single, multiple, or even a distributed computing resource. The ROM 206 may be any type of non-volatile memory, which may be programmable such as, mask programmable, flash, etc.

[0037] In addition, RAM 208 may be, for example, static, dynamic, synchronous, asynchronous, or any combination. The storage 210 may be Compact Disc (CD), Digital Versatile Disk (DVD), hard disks (HD), optical disks, tape, flash, memory sticks, video recorders, etc. While the display 220 might be, for example, a Cathode Ray Tube (CRT), Liquid Crystal Display (LCD), a projection system, Television (TV), etc. Audio 222 may be a monophonic, stereo, three dimensional sound card, etc.

[0038] The keyboard 224 may be a keyboard, a musical keyboard, a keypad, a series of switches, etc. The pointer 226, may be, for example, a mouse, a touchpad, a trackball, joystick, etc. While the I/O devices 228 may be a voice command input device, a thumbprint input device, a smart card slot, a Personal Computer Card (PC Card) interface, virtual reality accessories, etc., which may optionally connect via an input/output port 229 to other devices or systems. An example of a miscellaneous I/O device 228 would be a Musical Instrument Digital Interface (MIDI) card with the I/O port 229 connecting to the musical instrument(s).

[0039] The communications device 230 might be, for example, an Ethernet adapter for local area network (LAN) connections, a satellite connection, a settop box adapter, a Digital Subscriber Line (xDSL) adapter, a wireless modem, a conventional telephone modem, a direct telephone connection, a Hybrid-Fiber Coax (HFC) connection, cable modem, etc. While the external connection port 232 may provide for any interconnection, as needed, between a remote device and the bus system 202 through the communications device 230.

[0040] For example, the communications device 230 might be an Ethernet adapter, which is connected via the connection port 232 to, for example, an external DSL modem. Note that depending upon the actual implementation of a computer system, the computer system may include some, all, more, or a rearrangement of components in the block diagram. For example, a thin client might consist of a wireless hand held device that lacks, for example, a traditional keyboard. Thus, many variations on the system of FIG. 2 are possible.

[0041] Referring back to FIG. 1, the plurality of clients 108 are effectively connected to web sites, application service providers, search engines, and/or database resources represented by servers, such as the plurality of servers 104, via the network 102. The web browser and/or other applications are generally running on the plurality of clients 108, while information generally resides on the plurality of servers 104. For ease of

explanation, a single server 104, or a single client 108-1 will be considered to illustrate one embodiment of the present techniques. It will be readily apparent that such techniques can be easily applied to multiple clients, servers, or the like.

[0042] Referring now to FIG. 3, FIG. 3 depicts a block diagram of a multiprocessor environment computer system 300, which may be utilized by anyone of the clients or servers as depicted in FIG. 1. The multiprocessor computer system 300 includes a plurality of processors 302 (302-1, . . ., 302-N) coupled to a memory controller 310 via a bus 304. The memory controller 310 includes a memory 320 and is coupled to an I/O controller 330 via a second bus 312. The I/O controller 330 may include one or more devices 340 (340-1, 340-2 and 340-3) coupled via a bus 332.

[0043] In a computer system, such as depicted in FIG. 3, a user or system administrator may desire to load a trustable operating system. As referred to herein, the term trustable refers to the capability provided to either the user or a third party to later inspect the system 300 and determine whether a given operating system was loaded. Once determination of whether the given operating system was loaded, the user or third party may further determine whether the operating system has been loaded into a secure memory environment. This mechanism is not provided in conventional operating systems.

[0044] However, the described mechanism is of particular interest because it allows the trustable operating system to be loaded after untrusted software components have run. Moreover, this capability is robust in the presence of malicious software that may be attempting to compromise the computer system 300 from, for example, another processor in a multiprocessor system, while the system 300 may be attempting to register trustable software components.

[0045] As such, in a typical computer system, a processor 302 enforces privilege levels. These privilege levels restrict which system resources a particular software component may access. Accordingly, in a conventional system, such as depicted in FIG. 3, a user or third party is currently not available to load a trusted operating system via a unilateral request to one of the processors 302. As described in further detail below, a load (load secure region) instruction is provided in response to which load (load secure region) operation is performed to create a protected or secure memory environment in which a selected operating system may run with full privileges as a trusted operating system. Accordingly, as described herein reference is made, interchangeably, to a load instruction or a load secure region (LSR) instruction. In addition, reference is made, interchangeably, to a load operation or a load secure region (LSR) operation, performed in response to issuance of a load/LSR instruction.

[0046] Accordingly, FIG. 4 depicts a computer system 400 which includes the capability for unilaterally loading a trustable operating system within a secure memory environment. The mechanism provided is robust in the presence of malicious software that

may be attempting to compromise the computer system 400 from another processor of the multiprocessor system while the system is registering trustable software components. Accordingly, the computer system 400 is more or less configured as depicted in FIG. 4, however, in the embodiment described, the memory controller includes the memory protection element 402 as well as a third party verification block 450 as well as a memory hashing engine 470. Although provided in a specific embodiment, the computer system 400, as depicted in FIG. 4, represents an abstraction of a computer system utilized by any of the clients 108 or servers 104 as depicted in FIG. 1. However, those skilled in the art will appreciate that the distinction between the various components of the computer system 400 is a logical distinction only. In practice, any of the components may be integrated into the same silicon die divided into multiple die or a combination of both. In addition, the various components may be performed in microcode, software or device specific components.

[0047] As described herein, a certain memory region is considered to contain the trustable operating system which the user or system administrator desires to load. In one embodiment, this memory region contains a portion of the trustable operating system which executes with full privilege. As a result, the computer system 400 requires a mechanism for preventing untrusted software and devices from corrupting the secure memory environment. Accordingly, in a typical computer system, a mechanism is necessary for regulating processor or device memory transactions into the secure memory region.

[0048] In the embodiment depicted in FIG. 4, the memory controller 410 of the computer system 400 is responsible for forwarding a received memory transaction to the appropriate destination. However, in contrast to conventional computer systems, the memory controller 410 includes a memory protection element 420. In the embodiment described, the memory protection element 420 is a special structure which may block certain memory accesses from proceeding to memory 412. Writes that are blocked by the structure are discarded without affecting memory. In addition, depending on the system requirements, reads may return values in which all the bits are clear, in which the bits are all set, or random values.

[0049] Accordingly, the memory protection element may be utilized by initiation software in order to set up one or more memory regions as a secure memory environment described above. In addition, the initiation software may load desired software or an operating system within the designated memory regions of the secure memory environment. Once loaded, the software or operating system functions as secure software or a trustable operating system once the secure memory environment containing the software or operating system is created.

[0050] As a result, once the memory regions forming the secure memory environment are created, the memory protection element will protect the environment from

corruption. Accordingly, once the secure memory environment is created, the software or operating system contained therein is considered to be trustable. In addition to the memory protection elements 420, the computer system 400 includes an outside verification block 450. In one embodiment, the outside verification block is utilized in order to establish trust verification to an outside agent of the operating system or secure software contained within the secure memory region.

[0051] Accordingly, in one embodiment, an outside agent may establish trust or decide to trust the operating system contained within the secure memory environment by being provided the capability to inspect the software which resides in the secure memory environment. In one embodiment, rather than expose the entire region to inspection, the computer system 400 exposed a cryptographic hash value which represents the secure memory region. This value is generated by a memory hashing engine 470 after the region is secured and stored in a digest 460. In one embodiment, the cryptographic hash value of the secure memory environment represents a software identification value of the secure environment contained therein.

[0052] As such, once the secure software identification value is generated, the system 400 may store the secure software identification value in a digest 460. In one embodiment, the digest is a repository of information, which may be transmitted to an outside agent for the purposes of establishing trust. In the embodiment described, these values may be read and cryptographically signed by a hash signing engine 452 before being transmitted to the outside agent. As such, in the embodiment described, the digest 460 represents a bank of digest registers, which are accessed by the digest signing engine 452 via a secure channel 454.

[0053] Accordingly, the digest signing engine 452 utilizes the secure channel 454 to access the digest 460 and will sign the contents upon receiving a request to do so from an outside agent. By requesting such a signing, an outside agent may observe the stated components reported by the signed software identification value and decide whether or not to trust the computer system 400. In one embodiment, the digitally signed secure software identification value may be stored in a hardware token, which is accessible by the outside agent. This is provided in order to avoid software-to-software verification which may be easily circumvented in the presence of malicious software that may be attempting to compromise the computer system.

[0054] Referring now to FIG. 5A, FIG. 5A depicts secure memory environments 500 and 550 formed in accordance with an embodiment of the present invention. In one embodiment, the secure memory environments are formed in response to issuance of a load secure region instruction. In one embodiment, in response to issuance of the load secure region (LSR) instruction by security initiation software, a load secure region (LSR) operation is performed. As a result, the security initiation software will select software or

an operating system which it desires to become the secure software. Next, the secure software will direct the loading of the selected software within a designated memory region.

[0055] Once the software or operating system is loaded, the memory region within which the operating system is loaded will be designated by the initiation software, via an LSR instruction, to become the secure memory environment. Accordingly, the LSR instruction will be provided to a processor 402 (FIG. 4) which will execute an LSR operation directing the processor to unilaterally begin formation of the secure memory environment without cooperation from any of the other processors (402-2 through 402-4). In doing so, the processor 402 will establish an LSR exclusivity mutex to ensure that a current LSR instruction is not being performed.

[0056] As such, the processor will perform a series of actions, including verification that the initiation software is permitted to issue such an instruction (e.g., a privilege level check), followed by direction of the memory protection element to secure the designated secure memory environment. Once this is performed, the processor may collect information regarding the secure memory environment as well the identification as the initiating processor and record this information within the digest. As such, the processor will commit all values within the digest and direct the hashing engine to compute the secure software identification value, which will be stored in the digest and can be provided to an outside agent in order to establish verification.

[0057] Referring again to FIG. 5A, the initial software may request a secure memory environment 550 which includes a fixed base 554 and a fixed offset or entry point 552 in which each of the processors will be directed to following the issuance of a secure reset (SRESET), described in further detail below, operation once the secure memory region is formed. In addition, the secure memory environment 500 may include a variable base address 504 and a fixed offset entry point in which to direct each of the processors once the SRESET command is issued.

[0058] Accordingly, as described in further detail below, the SRESET command will force each of the processors to clear all outstanding transactions, memory information, tasks, code state and the like. In addition, the processors are directed to a specific entry point as depicted in FIG. 5A, such that the first operation performed following an SRESET is within the trustable operating system contained within the secure memory environment. In addition, at the time of SRESET, all processors in the system enter one of a plurality of known, privileged states, S_1, S_2, ... S_N.

[0059] In one embodiment, each of these states may be described by a set of parameters (e.g., initial privilege level, operating mode, register values, control values, etc.) that are either read from the platform at the time of SRESET (in which case, the values of the parameters are recorded, either directly or indirectly, in the digest log), or as part of the configuration of the processors. For example, in one embodiment, the privilege mode

of the processors following SRESET is fixed; it may be encoded in the processors' SRESET microcode sequences. However, the address at which the processor starts executing instructions following an SRESET operation may be fetched from the platform at the time of the SRESET.

[0060] In one embodiment, the data storage elements within the processor are set to constant values. Accordingly, the constant values may either be stored in, for example, a ucode ROM (e.g. STORE the value 0x55AA into register N). Alternatively, the consent values may be hardwired into the operation of the chip (e.g. upon receiving a RESET signal, all flip-flops in one section of the chip are reset/set). Finally, the values may be derived from algorithms expressed in the ucode ROM (e.g. for each address in a cache store the address of the element into the element).

[0061] In an alternate embodiment, data storage elements within the processors are set to specified parameter values. Accordingly, the parameters may be specified by the initiation software at the time of issuing the LSR instruction. These parameters may not be the exact values stored in the register. For example, a particular parameter may indicate, for example, "paging-enabled." The processor will know how to translate that parameter into STORE values A, B, and C into registers X,Y, and Z. At the time of issuing the LSR, these parameters are either (1) stored in a predetermined location within the platform (e.g. the digest, the chipset, or the initiating processor) or (2) transmitted to all processors. At the time of issuing an SRESET, then, each processor either (1) fetches the parameters from the storage location(s) or (2) already has its copy of all interesting parameters. The starting code address is one such parameter.

[0062] Consequently, as depicted in FIG. 5B, the secure memory region 602 will contain secured software/operating system code 610, as described above, which will be contrasted from the ordinary environment applications 640. As such, once the secure memory environment 602 is formed, the secure software 610 may either run to completion or initiate trusted applications (applets) 620, which may perform further functions as desired. Accordingly, as described above, the identity of the trustable operating system is recorded in the digest 460 and following an SRESET command, all processors will begin executing code belonging to that trustable operating system.

[0063] As a result, an outside entity may later request a signed version of the digest and be able to evaluate whether the system is in a trustable state. In one embodiment, the LSR and SRESET operations described above, may be exposed to initiating software in several ways, for example through I/O port accesses, memory accesses, instructions or the like. In one embodiment, the LSR operation is implemented as an instruction and the SRESET is implemented as a command to the platform chipset.

[0064] Accordingly, the LSR operation may be implemented in processor microcode. In addition, the SRESET command may be implemented by the chipset logic.

Furthermore, the processor resources may be used to implement the functionality of the memory hashing engine 470 (FIG. 4). That is, the memory hashing engine 470 may be a microcode sequence which leverages the compute sequences resources of the processor 402. In alternate embodiments, the memory hashing engine 470 may be implemented in either the memory controller 410 or I/O controller 430.

[0065] However, due to the fact that the LSR functionality is implemented in a processor, the platform will be able to distinguish between messages which are generated by the hashing engine 470 and those that are generated by general purpose initiation software. In one embodiment, a special message type is introduced, which is only available to the LSR memory hashing microcode. Consequently, macrocode software will be devoid of a mechanism to generate these types of messages. In one embodiment, the SRESET command may be a simple write to a certain I/O port address, which is decoded by the chipset. As a result, chipset then may take the actions described as the SRESET command.

[0066] While the memory protection element 420 is depicted as being integrated into the memory controller 410 as depicted in FIG. 4, those skilled in the art will appreciate that this mechanism can be implemented and distributed throughout the system 400 with the functionality appearing in the I/O controller and processors. Moreover, the digest and digest signing agent may be integrated into any of the chipset components described or may be implemented as separate platform devices. In the embodiment described, these components are illustrated as separate devices connected to the I/O controller 430. Procedural methods for implementing the teachings of the present invention are now described.

Operation

[0067] Referring now to FIG. 6, FIG. 6 depicts a flowchart illustrating a method 700 for unilaterally loading a secure operating system within a multiprocessor environment, for example as depicted in FIG. 4. At process block 710, it is determined whether a load secure region (LSR) instruction (load instruction) has been received. Once an LSR instruction is received, process block 712 is performed. At process block 712, it is determined whether a currently-active LSR operation (load operation) is detected. This check is performed because any of the processors may each receive an LSR instruction request from initiating software.

[0068] In other words, as described above, the mechanisms described herein provide a technique for the unilateral creation, by an initiating processor, to form a secure memory environment in response to, for example security initiation software. As such, when an LSR instruction is received, the receiving processor determines whether another processor has already received an LSR instruction which is currently being performed

(current LSR operation). Accordingly, when a current LSR operation is detected, the received LSR instruction will be discarded. Otherwise, process block 730 is performed.

[0069] Consequently, process block 730 is performed once it is verified that another processor is not performing an LSR instruction. At process block 730, the receiving processor will direct a memory protection element 420 to form a secure memory environment, as indicated by the LSR instruction. Finally, at process block 740, the processor will store a cryptographic hash value of the secure memory environment within a digest information repository 460. As described, the storing of the hash value, or secure software identification value, is utilized to enable security verification of the environment to an outside agent.

[0070] Referring now to FIG. 7, FIG. 7 depicts a flowchart illustrating an additional method 702 for initiation of an LSR instruction via security initiation software. At process block 704, the security initiation software will select one or more memory regions for inclusion as the secure memory environment. Once selected, at process block 706 the initiation software will direct a processor to load selected software or an operating system within a memory region of the one or more memory regions. Finally, at process block 708, the initiation software will issue an LSR instruction for formation of the secure memory region as the one or more memory regions to a selected processor. Once performed, control flow branches to process block 710 of FIG. 6.

[0071] Referring now to FIG. 8, FIG. 8 depicts a flowchart illustrating actions performed by a receiving processor in response to receipt of an LSR instruction from security initiation software. At process block 716, the processor will receive an LSR instruction from security initiation software. Once received, at process block 718, the processor will determine whether the security initiation software is authorized to issue the LSR instruction. When the initiation software is unauthorized, the LSR instruction is disregarded. Otherwise, at process block 720, the processor will determine if an LSR operation exclusivity mutex is detected. As described above, an exclusivity mutex enables an initiating processor to inform any of the other processors that an LSR instruction is currently being performed.

[0072] When an exclusivity mutex is detected, the received LSR instruction is disregarded. Otherwise, at process block 722, it is determined whether a secure reset (SRESET) command is detected. As will be described in further detail below, the SRESET command is utilized by the initiating processor to force each of the processors to accept the secure memory region/environment and begin performing operations within the operating system of the secure memory environment. When an SRESET command is detected, the received LSR instruction is disregarded. Otherwise, at process block 724, the processor will establish an LSR operation exclusivity mutex for the reasons described above. Once performed, control flow branches to process block 730, as depicted in FIG. 6.

[0073] Referring now to FIG. 9, FIG. 9 depicts an additional method 732 for directing a memory protection element 420 to form the secure memory environment of process block 730 as depicted in FIG. 6. At process block 734, the memory protection element 420 will clear the digest information repository. Once cleared, at process block 736, the memory protection element 420 will collect the secure memory environment information. In one embodiment, the secure memory environment information may include secure software identification information, processor versions, state information and the like. As such, the collection of the secure memory information provides registration of the software or operating system initially loaded by the initiation software. Finally, at process block 738, the memory protection element 420 will store the secure memory environment information within the digest information repository 460. Once stored, the software or operating system is registered as the secure software or secure operating system within the secure memory environment.

[0074] Referring now to FIG. 10, FIG. 10 depicts a flowchart illustrating an additional method 742 for generating the cryptographic hash value or secure software identification value of process block 740, as depicted in FIG. 6. At process block 744, the memory hashing engine 470 will hash contents of the secure memory environment to generate a cryptographic hash value as a secure software identification value. Next, at process block 746, the hashing engine 470 will store the secure software identification value within a hardware digest information repository 460. Once this is performed, the security memory environment, as well as the secure software, has been identified and registered. Accordingly, at process block 748, the processor will release the LSR operation exclusivity mutex to enable other processors within the system to perform an LSR instruction.

[0075] Referring now to FIG. 11, FIG. 11 depicts a method 750 for security verification of the secure memory environment to an outside agent. At process block 752, it is determined whether a security authentication request is received from an outside agent. Once received, at process block 754 the digest signing engine 452 will access the secure software identification value, via the secure channel 754, from the hardware digest 460. Once accessed, at process block 756 the digest signing engine 452 will digitally sign the secure software identification value. Finally, at process block 758 the signed identification value is transmitted to the outside agent.

[0076] Referring now to FIG. 12, FIG. 12 depicts a method 800 which illustrates the secure reset (SRESET) command, as described above, for example within the computer system 400 as depicted in FIG. 4. At process block 802, it is determined whether formation of the secure memory region is complete. In one embodiment, this may be determined by verifying whether information within the digest 460 has been committed following completion of an LSR instruction via an initiating processor. Once formation is

complete, process block 804 is performed. At process block 804, the initiating processor will issue a reset command to the plurality of processors 400 within the system 400. Once reset, at process block 820 the processor will enable read/write access to the one or more protected memory regions of the secure memory environment to each of the processors. As such, once read/write access is enabled, all processors will begin executing code belonging to the trustable operating system.

[0077] Referring now to FIG. 13, FIG. 13 depicts a flowchart illustrating an additional method 806 performed in response to the reset command of process block 804 as depicted in FIG. 12. At process block 808, the processor will issue to a system platform chipset, a processor reset command. Once issued, at process block 810 the chipset platform or the platform chipset will issue a reset request to each of the plurality of processors of the computer system 400. Once issued, at process block 812 each processor will clear all outstanding tasks, transactions, cache memory and the like such that outside actions do not result in, for example a system memory interrupt.

[0078] Finally, referring to FIG. 14, FIG. 14 depicts a flowchart illustrating an additional method 822 for enabling processor read/write access of process block 820, as depicted in FIG. 12. At process block 824, the processor will commit all information within the digest information repository 460 to indicate an active SRESET command. Once committed, at process block 826 each of the plurality of processors 402 within the system 400 will be placed within a known privilege state or high privilege state available in the computer system 400.

[0079] Next, at process block 828 each processor will be directed to a known state which may include, for example, a one of a plurality of known privileged states (S_1, S_2, . . ., S_n), an operating mode, register and control values as well as a predetermined entry point within the secure memory region, for example as depicted in FIG. 5A. Finally, at process block 830, a memory protection element 420 will be directed to re-enable processor read/write access to the secure memory environment. In this way, the identity of the trustable operating system is recorded in the digest and following an SRESET, all processors will begin executing codes belonging to that trustable operating system. Moreover, an outside agent may later request a signed version of the digest and be able to evaluate whether the system is in a trustable state.

Alternate Embodiments

[0080] Several aspects of one implementation of the secure memory environment multiprocessor system for providing unilateral loading of a secure operating system within a multiprocessor environment have been described. However, various implementations of the secure memory environment multiprocessor system provide numerous features including, complementing, supplementing, and/or replacing the features described above. Features can be implemented as part of the same silicon, multiple die, a combination of both

or as part of the microcode, or as specific hardware or software components in different implementations. In addition, the foregoing description, for purposes of explanation, used specific nomenclature to provide a thorough understanding of the invention. However, it will be apparent to one skilled in the art that the specific details are not required in order to practice the invention.

[0081] In addition, although an embodiment described herein is directed to a secure memory environment multiprocessor system, it will be appreciated by those skilled in the art that the teaching of the present invention can be applied to other systems. In fact, systems for unilaterally loading a verifiable trustable operating system are within the teachings of the present invention, without departing from the scope and spirit of the present invention. The embodiments described above were chosen and described in order to best explain the principles of the invention and its practical applications. These embodiment were chosen to thereby enable others skilled in the art to best utilize the invention and various embodiments with various modifications as are suited to the particular use contemplated.

[0082] It is to be understood that even though numerous characteristics and advantages of various embodiments of the present invention have been set forth in the foregoing description, together with details of the structure and function of various embodiments of the invention, this disclosure is illustrative only. In some cases, certain subassemblies are only described in detail with one such embodiment. Nevertheless, it is recognized and intended that such subassemblies may be used in other embodiments of the invention. Changes may be made in detail, especially matters of structure and management of parts within the principles of the present invention to the full extent indicated by the broad general meaning of the terms in which the appended claims are expressed.

[0083] The present invention provides many advantages over known techniques. The present invention includes the ability to unilaterally load a secure operating system within a multiprocessor environment. The present invention provides a mechanism by which a system may install a trustable operating system within a secure memory environment. Accordingly, a user or system administrator may utilize an LSR instruction followed by an SRESET instruction, as described above, to load a trustable operating system. Once completed, an outside agent may later inspect the system and determine whether a given operating system was loaded and if so, if it had been loaded into a secure environment or not. Consequently, this described mechanism is of particular interest because it allows the trustable operating system to be loaded after untrusted software components have run. Further, the mechanism is robust in the presence of malicious software that may be attempting to compromise the computer system from another processor in a multiprocessor system while the system is registering trustable software components.

[0084] Having disclosed exemplary embodiments and the best mode, modifications and variations may be made to the disclosed embodiments while remaining within the scope of the invention as defined by the following claims.

Page 19 of 20